AFRL-IF-RS-TR-1999-241
Final Technical Report
November 1999

BRIDGING THE DEVELOPMENT GAP

Mercury Computer Systems, Inc.

Sponsored by
Advanced Research Projects Agency
DARPA Order No. D351

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1999-241 has been reviewed and is approved for publication.

APPROVED:

Ralph Kohler
Project Engineer

FOR THE DIRECTOR:

Northrup Fowler
Technical Advisor
Information Technology Division

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTC, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

BRIDGING THE DEVELOPMENT GAP

Craig Lund

Contractor: Mercury Computer Systems, Inc.
Contract Number: F30602-95-2-0037
Effective Date of Contract: 28 September 1995
Contract Expiration Date: 30 September 1997
Short Title of Work: Bridging the Development Gap
Period of Work Covered: Sep 95 - Sep 97

Principal Investigator: Craig Lund
Phone: (508) 256-1300
AFRL Project Engineer: Ralph Kohler
Phone: (315) 330-2016

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by Ralph Kohler, AFRL/IFTC, 525 Brooks Road, Rome, NY.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	NOVEMBER 1999	Final Sep 95 - Sep 97

4. TITLE AND SUBTITLE
BRIDGING THE DEVELOPMENT GAP

5. FUNDING NUMBERS
C - F30602-95-2-0037
PE - 62301E
PR - D002
TA - 01
WU - P3

6. AUTHOR(S)
Craig Lund

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Mercury Computer Systems, Inc.
199 Riverneck Road
Chelmsford MA 01824-2820

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

Air Force Research Laboratory/IFTC
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-IF-RS-TR-1999-241

11. SUPPLEMENTARY NOTES
Air Force Research Laboratory Project Engineer:Ralph Kohler/IFTC/(315) 330-2016

12a. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Bridging the Development Gap is contractual cooperative agreement between Mercury Computer Systems, Inc. and DARPA. This program was developed because a software gap exists between the workstation-based research phase of a signal processing project and the more contained prototyping phase. The transition requires a shift from a workstations rich environment into an embedded system that typically offers only basic system software. The gap reflects more than just a lack of software tools. It concerns new challenges such as: parallel decomposition, optimizing data transfer, heterogeneous processing, interfacing with I/O devices, memory constraints, as well as real-time throughput and latency challenges.

Mercury has bridged the indicated software gap by delivering on this program a deployment-focused environment for algorithms created in a popular research language, MATLAB (and its companion SIMULINK). The project has had the full cooperation of The Math Works, owner of MATLAB and SIMULINK. Mercury's discussions with Prime Contractors building large, embedded systems had shown MATLAB to be nearly universal tool of choice within the research phase of these projects. Demand for a MATLAB deployment path thus clearly existed.

The most significant element required to pull MATLAB and SIMULINK into parallel processing is to create a "mapping tool" and an underlying "component" run-time system.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Computer Network Security, Intrusion Detection	72
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Standard Form 298 (Rev. 2-89) (EG)
Prescribed by ANSI Std. 239.18
Designed using Perform Pro, WHS/DIOR, Oct 94

Table of Contents

Appendixes

List of Tables

List of Figures

1.0 Executive Summary

This final report summarizes the results of cooperative research, that Mercury Computer Systems, Inc. performed on the "Bridging the Development Gap" program. The program was supported by the Defense Advanced Research Projects Agency (DARPA) under BAA95-19, entitled "Programming and Runtime Environments and Operating Systems". The high-level program goals as described in Mercury's original proposal were:

- Mercury intends to bridge the software gap between defense research, prototypes, and deployment development stages. Developers who take advantage of Mercury's proposed innovations can expect experience increased productivity resulting in better solutions sooner, at lower cost.
- Mercury's proposed component run-time system moves important functionality away from programming tools and into system software (where the functionality belongs). Mercury will work to make its run-time interface an open standard that is widely supported by tool and embedded system vendors. Improved interoperability will result.
- With improved interoperability, the overall embedded community will gain from a larger collection of software tools, each supporting multiple hardware platforms.
- With improved interoperability, defense tool vendors can focus their limited resources on building better tools, instead of porting into different operating systems.
- The underlying component programming model Mercury advocates promotes the re-use of software modules and maintainability of large software projects.

A software gap exists between the workstation-based research phase of a signal processing project and the more constrained prototyping phase. This transition requires a shift from a workstation's rich environment into an embedded system that typically offers only basic system software. The gap reflects more than just a lack of software tools. It concerns new challenges such as: parallel decomposition, optimizing data transfer, heterogeneous processing, interfacing with I/O devices, memory constraints, as well as real-time throughput and latency challenges.

Mercury has bridged the indicated software gap by delivering on this program a deployment-focused environment for algorithms created in a popular research language, MATLAB (and its companion SIMULINK). The project has had the full cooperation of The MathWorks, owner of MATLAB and SIMULINK. Mercury's discussions with Prime Contractors building large, embedded systems had shown MATLAB to be a nearly universal tool of choice within the research phase of these projects. Demand for a MATLAB deployment path thus clearly existed. On their own, several primes had already undertaken projects to provide rudimentary interfaces between Mercury's embedded platform and MATLAB.

The most significant element required to pull MATLAB and SIMULINK into parallel processing is to create a "mapping tool" and an underlying "component" run-time system. Mercury has delivered functionality that is not specific to MATLAB nor SIMULINK, but can be leveraged by other tool vendors. To help achieve this broad goal, Mercury and The MathWorks have fully documented all interfaces. Mercury strongly believes that our industry needs those open interfaces that facilitate interoperability between development tools.

Mapping Tool.---Most of the efforts for this program focused on the Mapping Tool. This is because today's SIMULINK cannot generate code for multiprocessor environments. Our Mapping Tool pulls SIMULINK into the parallel world. SIMULINK visually represents applications as a graph of interconnected functions in boxes connected by lines. The lines represent data flow between functions.

Mercury's Mapping Tool enables manual assignment of functions to specific hardware and assignment of interconnections to specific data transfer APIs. Proper mapping (assignment) is critical to meeting embedded system performance and efficiency requirements. To permit manual assignment, the Mapping Tool's graphical user interface simultaneously shows a SIMULINK's logical "netlist" and the target hardware's physical reality. Therefore, our Mapping Tool supports graphical specification of the physical configuration of embedded systems.

Component Run-Time System.---Mercury has developed an underlying run-time system that supports a component programming model. Such a run-time system processes a "netlist" which specifies the interconnection and processor assignments of software modules available as object code. From the netlist, the run-time system synthesizes the required executable images, loads the images into appropriate processors, sets up the "interconnections" as inter-process communication objects, and begins execution of the application.

The underlying "netlist" specification is actually a scripting language. Specifically, we have created a specialized Tool command language (Tcl) extension package that we call ACL (Application Configuration Language).

Mercury's "Bridging the Development Gap" program started in August, 1995. At its own expense, Mercury had already started work on a standard component runtime system (Talaris, see Appendix A). Mercury had delivered review copies of a detailed Talaris interface specification to several major software tool developers, including The Math Works.

We have completed the software tools as proposed for the program in FY97. Our test partners, MITRE, NUWC, and Integrated Sensors Inc., have validated anticipated productivity benefits. The program ended in September 1997. Two other DARPA-sponsored programs have build upon our results in FY98 and beyond.

Multiple commercialization programs exist, mostly at industry's expense. Spectron Microsystems has planned a commercial variant for their SPOX-MP operating system. The component runtime system created under the program is now in commercial use by the CapCASE visual development environment (see Appendix B) from Matra Cap Systemes (France). UCB's Ptolemy group, led by Professor Edward Lee, is building our runtime into Ptolemy in conjunction with new research into scaleable systems.

2.0 Introduction

This final report describes recent research and development work related to BAA95-19 that has significantly improved developer productivity for parallel programming of signal processing applications today, while laying the groundwork for dramatic advances in the future.

2.1 Application Markets of Interest

Mercury builds computers primarily for embedded applications that process live sensor data. In the government electronics area, Mercury RACE® systems fit into radar, sonar, and signal intelligence systems. For the diagnostic medical imaging market, Mercury products connect directly to scanners for magnetic resonance imaging (MRI), computed tomography (CT), positron emission tomography (PET), and digital X-ray. Emerging application markets such as digital video and wireless communication processing are expanding opportunities for multicomputers into areas that require increasing bandwidth capacity.

These applications are also at the forefront of research. New sensors, new algorithms, and new technology continually push what is possible, and more importantly for Mercury, what is required from the computing environment. Our customers depend on "rapid prototyping" and implementation – flowing results from research to product as quickly as possible. The life-cycle stages of a typical embedded application are represented in Figure 1.

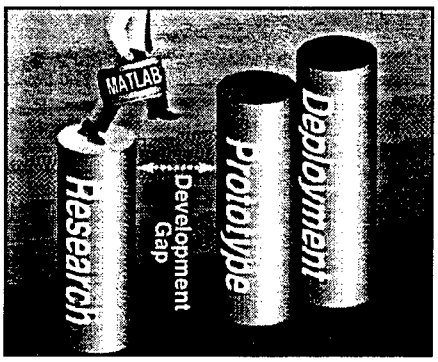


Figure 1. Life cycle of typical embedded application.

The cylinders in Figure 1 represent the steps a new algorithm typically goes through between inspiration and volume deployment. We have labeled the steps "research," "prototype," and "deployment." The prototype and deployment phases require a real-time architecture capable of connecting directly to real-time streams of high-bandwidth data. The deployment phase in particular runs on a real-time target, not on the host workstation.

2.2 Importance of MATLAB

A survey of Mercury's customers has shown that a significant majority of algorithms deployed on our systems began their life cycle on a workstation in the MATLAB® programming environment from The MathWorks. This high-level tool enables the researcher to conceive and explore algorithms easily.

The MathWorks has also added tools to the MATLAB product family to address the transition from research to prototype. These tools include the MATLAB Compiler to translate MATLAB M-files to C source; the MATLAB C Math Library for running that C code independent of MATLAB itself; the SIMULINK® block-diagram environment for simulating controls, signal processing, and other data-flow systems; and the Real-Time Workshop for generating C code from SIMULINK models.

However, the gap to deployment remains. The MATLAB C Math Library runs on the host, so it does not address deployed target-based implementations. Also, the MathWorks tools do not address the issues of scaling to multicomputer targets.

3.0 Bridging the Development Gap

To bridge the gap from the research to real-time implementation, two things will have to be done. First, the MATLAB C Math Library must be ported to the target environment. Second, a mechanism must be created to define and implement a scalable solution. This latter point will build on proven component programming concepts developed in other markets.

While it is not necessary to use component programming techniques to leverage the embedded MATLAB C Math Library, a few component programming basics are presented in the next section, followed by a description of the embeddable RACE MATLAB environment. This is followed by an overview of the component programming infrastructure. Finally, we look at performance issues and future developments.

3.1 Some Component Programming Concepts

In component programming, a software application is expressed as an interconnection of software Modules that executes on a configuration of hardware Modules.

A software Module consists of executable code that operates on data and commands via one or more Ports of the Modules. Interconnections of Ports between Modules are Connections. Graphically, the relationships of Modules, Ports, and Connections are shown in Figure 2.

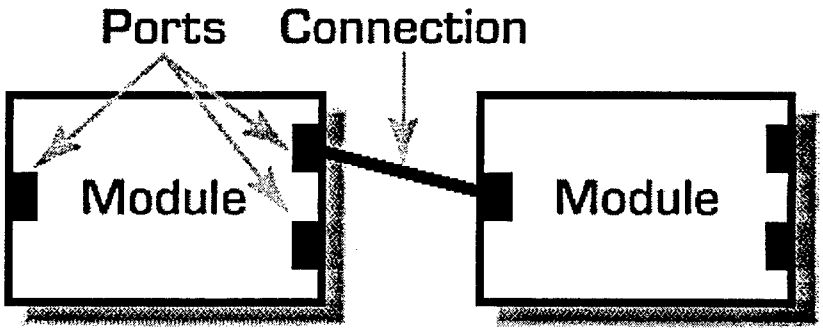


Figure 2. Representation of Modules, Ports and Connections

In the RACE implementation, Modules are POSIX threads or processes, Ports are various types of protocols (e.g., message passing, synchronization, and shared-memory application programming interfaces (APIs)), and Connections are objects that attach to Ports.

Hardware Modules consist of processors and their memory systems, the interface to the processor, and the connection of interfaces (e.g., connection to a shared bus or point-to-point fabric). Component programming is analogous to the ECAD design principles of Part, Pin, and Signal, as shown in Table 1.

Table 1. ECAD analogy components for building electronics compared to multicomputing component definitions

ECAD	Software	System Hardware
Part	Module	Processor
Pin	Port	Interface
Signal	Connection	Connection

Just as ASIC designers have leveraged reusable component methodology for rapidly creating complex chips, multicomputing application developers will also reap productivity gains by using methods and tools that leverage component technology. A complete application consists of software components and their Connections, system hardware components and their Connections, and the assignment (or mapping) of the software components to the system hardware components. An example is given in the next section.

3.2 Component Application Example

When thinking about a programming problem, a signal processing engineer usually draws a block diagram like the one in Figure 3. For applications characterized by a series of transformations, such as in a myriad of signal, image, and media processing applications, sketches of the type in Figure 3 are the most "natural" manner in which the application engineer expresses the application.

In Figure 3, the blocks represent software components written in MATLAB, C, assembly language, or whatever is most appropriate; the lines show Connections, or how Modules communicate with shared memory and semaphores, and other techniques. The coders of

the individual software Modules can create reusable Modules without extensive knowledge of the intricacies of the total application or the nuances of the target operating system.

But note, Figure 3 shows only the software view and does not reflect any specifics of the hardware upon which ultimately the algorithms will run.

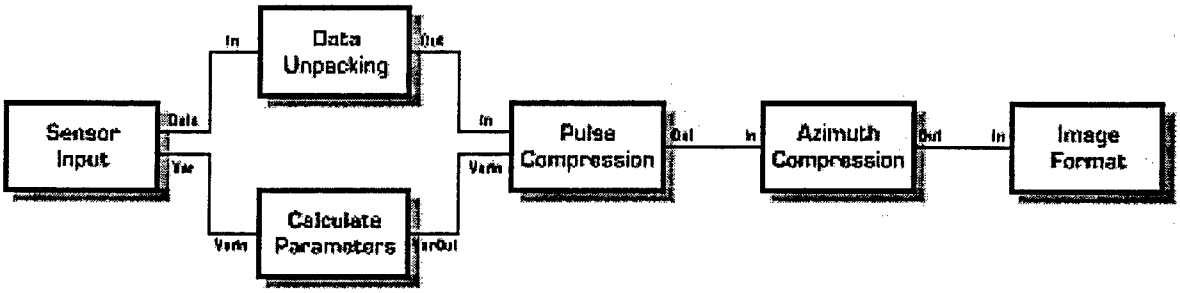


Figure 3. A software model of a typical Synthetic Aperture Radar (SAR) application shown as a collection of interconnected software modules.

In Mercury's heterogeneous RACE architecture, target processors include i860, PowerPC™ , and SHARC® DSPs. A small configuration appears in Figure 4.

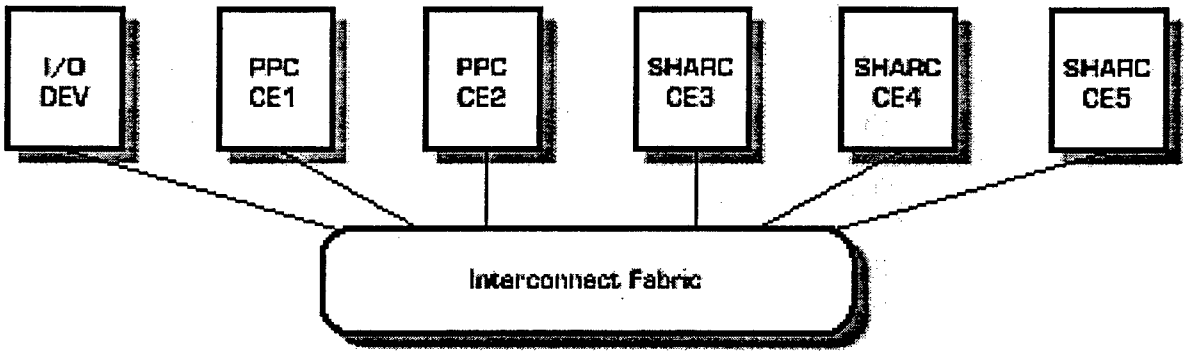


Figure 4. A simple hardware configuration.

We can consider an inventory of Modules compiled for those processors as a set of reusable software components.

In its simplest form, using component programming techniques for multicomputing is to execute each of the software Modules in Figure 3 in parallel on the hardware in Figure 4. Simply assign each block to its own processor, as in Figure 5. If this assignment does not produce the desired throughput, then the engineer may decide to parallelize a single Module across multiple processors (Figure 6) to improve the overall performance.

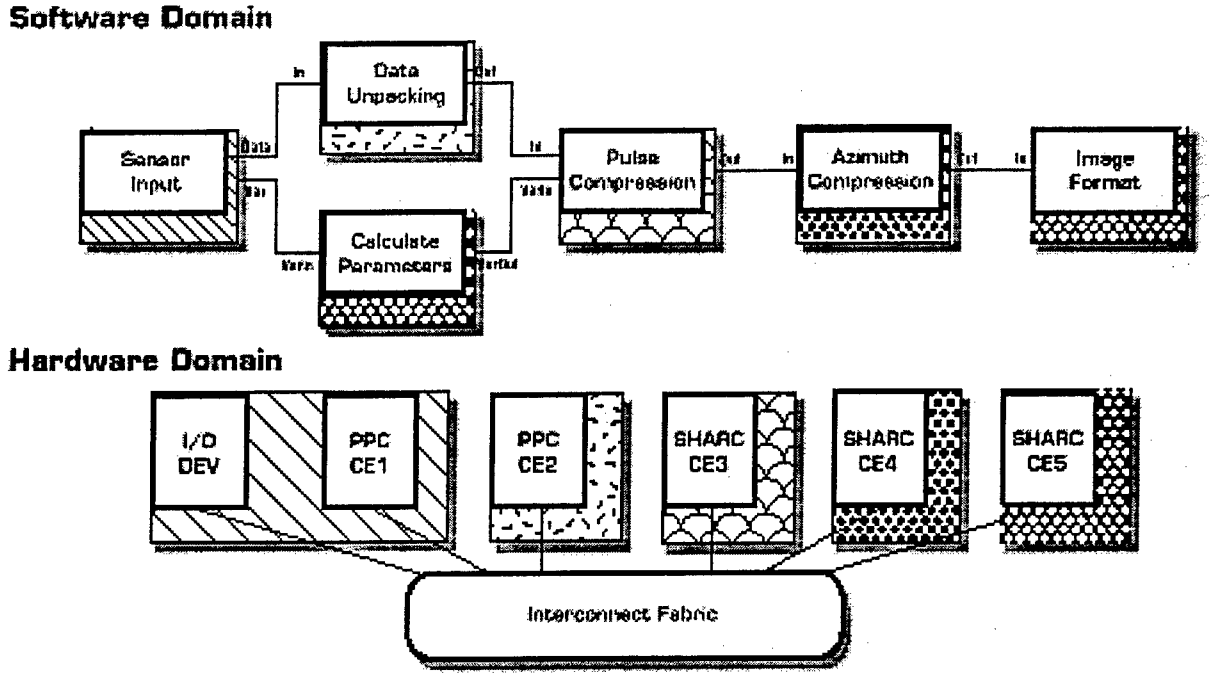


Figure 5. The illustration shows assignment from the software domain to the hardware domain. The fill pattern indicates one example of how the software model is assigned to the hardware model.

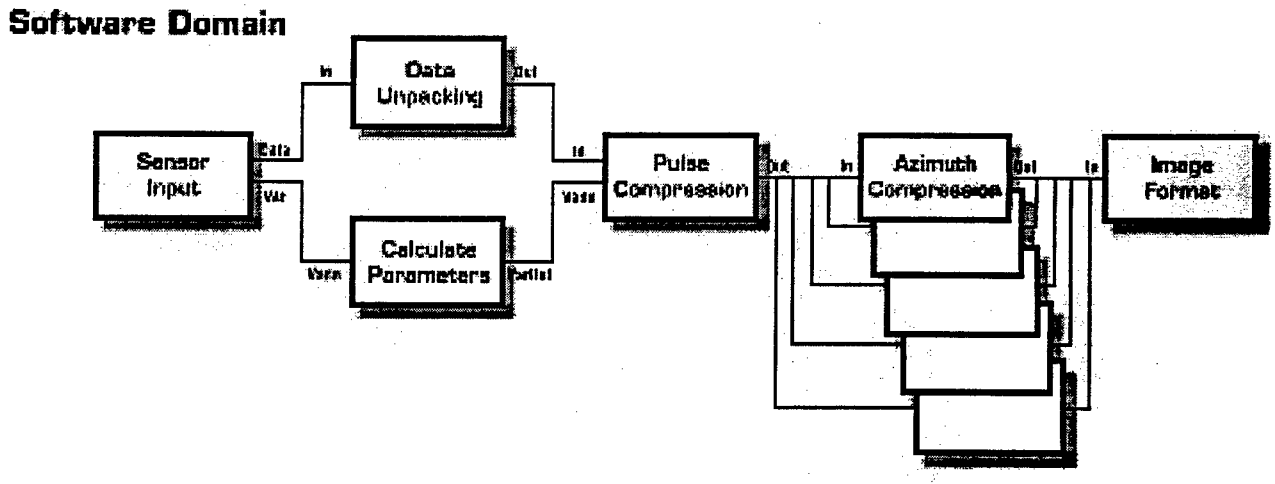


Figure 6. In this example, a scalable function is created and assigned to multiple processors to improve performance.

Mercury has created an environment that implements the thought process represented in Figures 3 to 6, from specification through execution. This environment in part relies on the Application Configuration Language (ACL). For ACL overview materials, reference, and tutorial, see:

- www.mc.com/talaris_fold/talariseet.html,
- www.mc.com/backgrounder_folder/icassp/icassp.html,
- www.mc.com/talaris_fold/talaris/slide0.html.

But before we can describe the component programming environment, we will describe how to turn MATLAB M-files into components.

4.0 MATLAB on RACE

The MathWorks and Mercury have collaborated to accomplish the task of porting the MATLAB C Math Library to target embedded processors. A developer, using the MATLAB compiler (mcc) and the RACE tool-chain, can compile an M-file to an object file that is linked with the MATLAB C Math Library and other libraries. Since MATLAB Modules can call C entry points, all of the APIs provided in RACE and by third parties are available to the MATLAB developer.

Assuming that a "monolithic" M-file exists for an application, the following steps are required to take advantage of the component programming tools for MATLAB:

1. The monolithic M-file is carefully studied and re-implemented as multiple separate M-files that each represent a piece of useful processing. The M-files will become individual software Modules.
2. Each M-file is compiled with the MATLAB compiler to create a C code version.
3. An ACL template is created to describe the Port interface for each M-file. This is typically a few lines, very similar to a C prototype declaration, that lists the input and output Ports, plus any attributes and properties of each Port. Port types for the first implementation are limited to MATLAB matrices and synchronization (via semaphores). This step is easily accomplished with a text editor or can be semi-automatically generated using the Inspector tool described below.
4. The ACL template files and the compiled M-files are input to a utility program that creates a C code "Port wrapper" around each compiled M-file. The output of the utility program is passed on to the C compiler to produce object files that represent reusable MATLAB software Modules.
5. Typically, but not always, object files would be organized as libraries using a standard archiving utility.

4.1 The Talaris Environment for Component Programming

A workstation hosts a collection of tools, the Talaris Modeler, and a variety of generators. The output of a generator is a "Launch Kit" for a specific target platform. A Launch Kit contains all the necessary image files and data to load, initialize, and execute the application. A small Launcher program is required on the target platform to open a Launch Kit, and perform the launching (load, initialize, and execute).

With this infrastructure, application development is equated to building a fully specified and populated application model in the Talaris Modeler. A fully specified application model contains:

- A system hardware model that expresses the instances of hardware Modules and their interconnection.
- A software model that expresses the instances of software Modules and their interconnection.
- The assignment of software Modules to hardware Modules.

A fully populated model means that object files (i.e., a ".o" file or library entry) exist for each software Module (for the assigned hardware Module type) and the hardware exists. Many useful development activities can be accomplished without a fully populated model, but that is not a subject of this report.

Figure 7 represents the component programming infrastructure developed as a result of Mercury's ongoing research and development efforts, with assistance from DARPA (BAA95-19). This report provides an initial description of this environment.

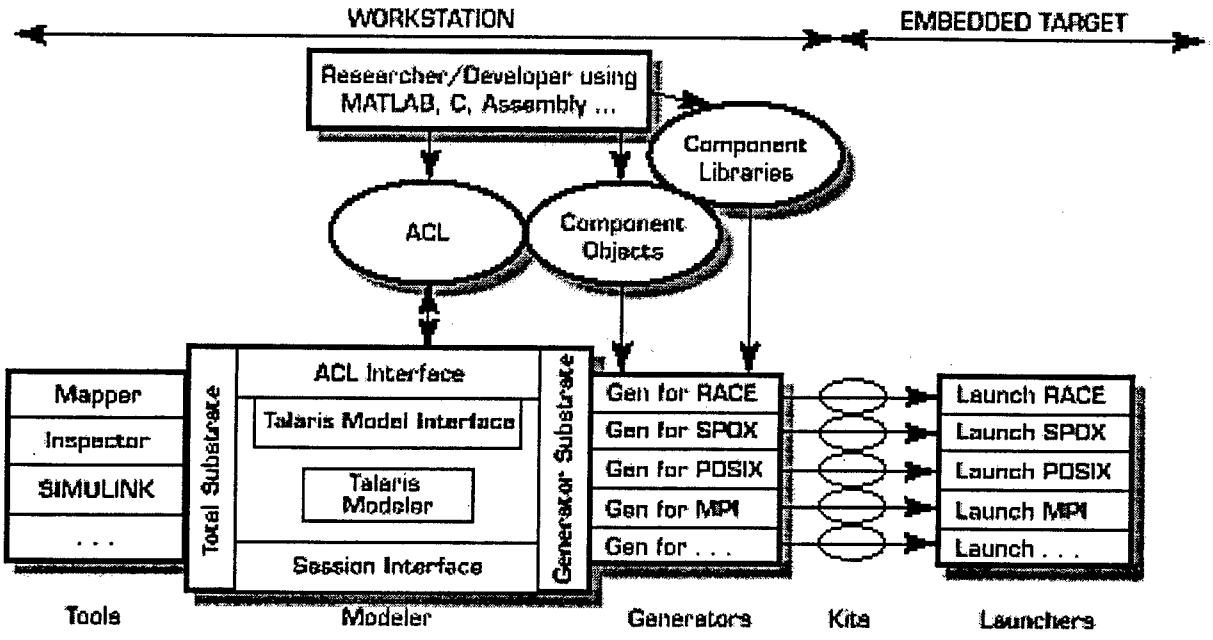


Figure 7. Talaris environment as a framework for component programming.

Current tools in Figure 7, future tools, and the ACL are means by which the user builds the model components. The tool substrate of Figure 7 is designed to allow simultaneous interaction with the Talaris Modeler for multiple tools. The current tools are focused on expressing the application model. We will discuss other types of tools in Section 6, Summary and Future Research. A brief description of the current application model expression methods is:

- **The Inspector Tool**--Inspector is a browser-like graphical user interface (GUI) that shows all class types and instances of all Module types. Properties and attributes of all objects can be inspected and modified. New Module types and instances can be created with Inspector.

- **The Mapper Tool**--Mapper is a browser-like GUI that shows the various Talaris domains and assignments between the domains. Domains are created, in part, to facilitate the assignment problem of scalable applications. The current four domains are software, process, target, and hardware. Individual software Modules are first mapped into processes. Next, processes are mapped onto an idealized hardware configuration (target domain). Last, the ideal hardware configuration is mapped onto the actual hardware that a user has available at that moment. The reader is referred to the ACL references in Section 3.2. Mapper can be used to make, modify, or view assignments. Making assignments across domains is done with a "click-and-drag" interface.

- **SIMULINK**--For our current research, Mercury and The MathWorks used the diagrammatic GUI of SIMULINK for "box-and-line" representation of software and hardware models. The reader should note that this use of Simulink is strictly as a drawing editor GUI and has no other functional relationship to The MathWorks Simulink product. The user can create diagrams of software or hardware models which are then translated to the Talaris Modeler. No assignments are done in the SIMULINK GUI; typically, assignments are done with Mapper.

- **ACL**--The Talaris Modeler contains an ACL interface for importing ACL programs. ACL programs can express a complete or partial application model. As with all the tools described here, the application model can be built incrementally; in the ACL case, by importing a series of ACL programs. The Talaris Modeler can also export ACL so that any changes done to the model can be captured in ACL. Use of ACL by the application engineer is optional, and no ACL knowledge is required to use the tools.

The Talaris Modeler offers completeness checks as the application model evolves. When fully specified, assigned, and populated, the model is ready for kit generation. Currently a generator exists for Mercury RACE systems, and the Spectron SPOX-MP operating system environment. The other generators shown in Figure 7 are under consideration for future work.

5.0 Application Performance

A goal of the Talaris component programming research is to maintain performance while gaining the productivity and portability benefits of component methodology. The Talaris Modeler does not add any runtime code nor perform any runtime orchestration of Modules.

The Launcher does perform initialization sequences (e.g., initialization of interprocess communication objects such as sockets, semaphores, and mapped memory areas) that are not expressed in Modules but are derived from the application model and specified in the Launch Kit. Such initialization actions are not considered part of the actual running application.

Since these initialization sequences can be quite tedious, error-prone and vendor-specific derived initialization is a significant productivity benefit of the component programming approach. A Talaris Generator builds an executable as specified by the application

model. If the model expresses what the developer would normally do manually, then execution time difference between manual methods and Talaris generation is nil.

Actual performance depends on:

1. The efficiency of the software Modules for each type of processor,
2. The implementation of the various Port and Connection types for the target platform, and
3. The effectiveness of the software-to-hardware assignment.

The first issue is dependent on the writer of the software Module, and for high-level language Modules, the quality of the compiler. The second dependency is the responsibility of the platform vendor or possibly a third-party API implementation. Finally, the last point above is currently in the realm of the application engineer who must empirically or by other means develop an optimum assignment.

ACL and the current tools are present to help the developer build application models with perhaps thousands of software Modules distributed across hundreds of processors. Future research offers advancement for issues that go beyond application building to further boost development productivity.

6.0 Summary and Future Research

Mercury has created an environment that implements the DSP and data-flow thought process from specification through execution. A core modeling tool has been developed with which other tools can interact. Application experts prefer this environment because it matches how they were trained to think about signal processing problems.

Scaling an application from a small laboratory hardware configuration to a larger deployed configuration can be simplified using this methodology. With this approach, the Talaris Modeler infrastructure delivers its significant productivity benefits without adding any appreciable performance overhead at runtime.

A summary of the results gained from our efforts on BAA95-19 is shown in Table 2.

Table 2. Summary of BAA95-19 results.

ACTIVITY	BEFORE BAA95-19	AFTER BAA95-19
Algorithm design and test	MATLAB on workstation	MATLAB on workstation
Reusable component design	Embedded in doc and code	Captured in ACL
Create software components	Manual code development	MATLAB Compiler Talaris Wrapper Tool
Connect components	Hand-coded variable names	Talaris Inspector Tool SIMULINK Config. Toolbox
Map software to hardware	Hand-coded initialization	Talaris Mapper Tool
Build	Makefiles and shell scripts	Talaris Generator
Run	Shell scripts and setup code	Talaris Launcher

Unlike visual tool developments of the past, the focus of this stage of the research was applying component programming constructs to – and developing a modeler for – multicomputing. The substrate that Mercury developed for Talaris uses component programming as a way to build, maintain, and update a model of the application. Figure 8 illustrates the conceptual model of our activity today and possible future directions. The modeler holds a dynamic model of an application so that various tools interact with the model, sometimes simultaneously, to scan, modify, or annotate the model as an application migrates from a functional specification to an optimized running application.

Future development might include simulation, performance analysis, automatic assignment, and fault reconfiguration tools. As a simple example of interaction among tools, consider an iterative cycle between an assignment tool and a performance analysis tool. Given a running application, the performance analysis tool updates the model (that is, modifies properties of Modules) with new performance metrics. The assignment tool

reads the new metrics and reassigns Modules for an improved optimization. Similar interactions, through the application model, are anticipated for the other tools in Figure 8.

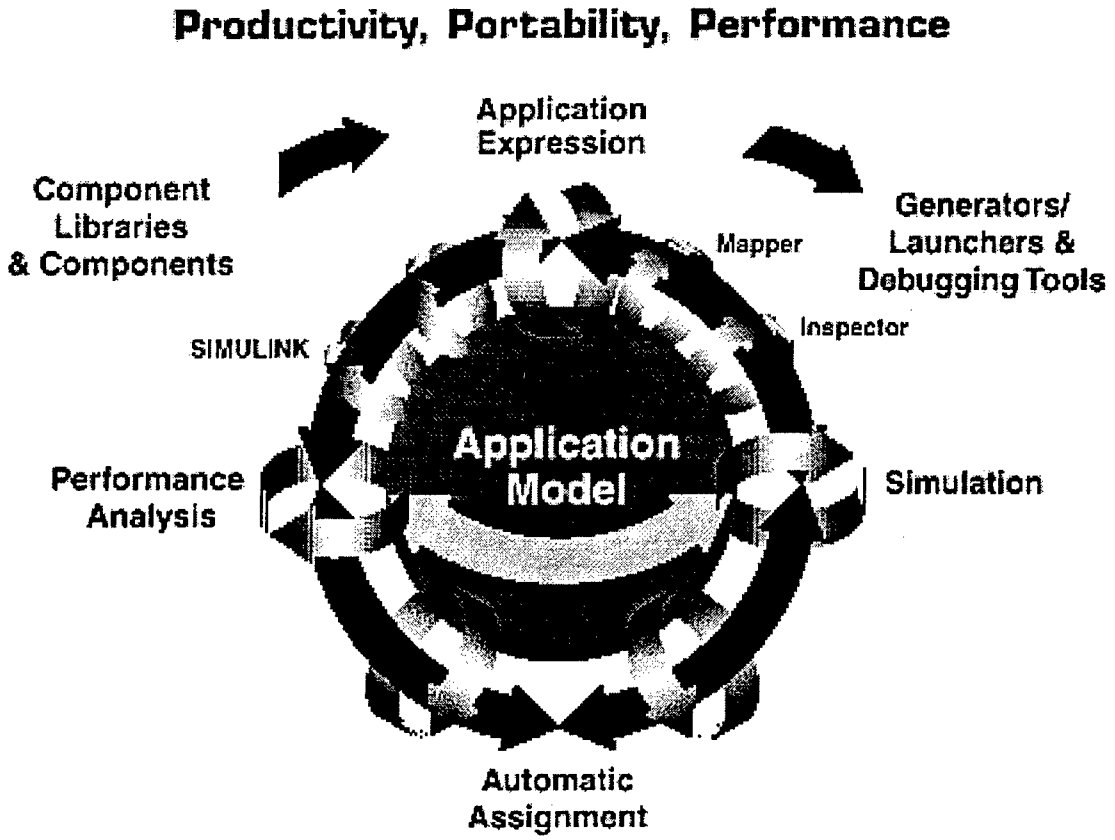


Figure 8. A function view of the Talaris Modeler environment with future tool examples.

The Talaris Modeler uses open, documented interfaces incorporating Java and the Tool Command Language, Tcl, and is also platform-independent. Our plans include generators for other computer architectures and integration of other types of advanced tools.

Our program has made significant contributions towards the objectives outlined within BAA95-19 and has produced the following benefits:

- Resulted in a "parallel, embedded, heterogeneous, real-time" MATLAB and bridged the software gap between research, prototypes, and deployment. Developers who take advantage of our innovations will experience increased productivity resulting in better time-to-solution at less cost (see Appendix C).
- Our component run-time system has moved important functionality away from programming tools and into system software (where the functionality belongs). We believe our run-time interface can become a standard that could be widely supported by tool and embedded system vendors.
- As a result of improved interoperability, the overall embedded community gains from a larger collection of software tools, each supporting multiple hardware platforms.
- As a result of improved interoperability, tool vendors can focus their limited resources on building better tools, instead of porting into different operating systems.
- The underlying component programming model we advocate promotes the re-use of software modules and maintainability of large software projects.
- Our documented interface descriptions can become the basis of an industry-wide standardization effort.

The research performed on this program has sought to eliminate significant steps from the development process used in most real-time, embedded, parallel processing projects. Therefore, our partnership with DARPA and Rome Laboratory has contributed to the United States' overall goal of maintaining a technological and competitive edge in the world. Our partnership has done this by making it faster and easier to deploy high performance computing technologies in typical embedded signal and image processing applications.

Appendixes

[A] **Talaris**— The application framework for scalable heterogeneous systems (Presentation)

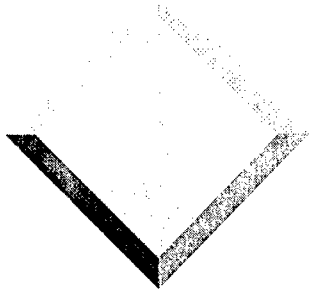
[B] Talaris applied to **Peakware** product (Mercury product announcement)

[C] Further product developments towards **MATLAB** use (Mercury product annoncement)

Appendix A.

Talaris— The application framework for scalable heterogeneous systems

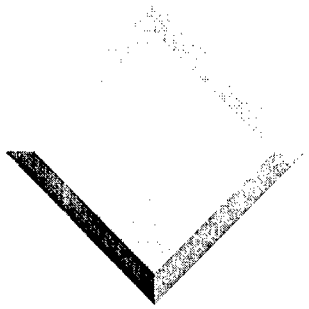
(Presentation)



Talaris

The Application Framework for Scalable Heterogeneous Systems

Mercury Computer Systems, Inc.



Observations

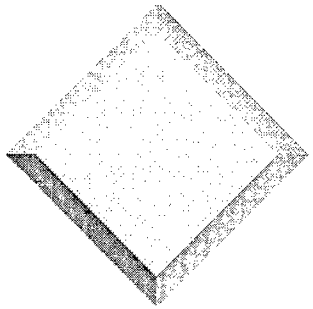
What we know

system size and raw CPU power is just the beginning...
hetero architectures offer new opportunities, but also complexity
application-specific configurations are much more competitive
developers know their application software, the rest is "overhead"
high-level tools tend to be specialized and expensive

- each targets specific applications and design methodologies
- each is coded and optimized for platform(s)

Key factors

the different needs of developers and tools
neutrality with respect to third-party developers
open, expandable, portable
the special needs of large / complex applications
concrete, technically feasible, viable for real applications
...and worth the investment



Conclusions

Applications are not just source code anymore

- configuration information should not be in source code
- makefiles and shells scripts are woefully inadequate

Provide an application framework for both people and tools

The Eight Challenges

1. Centralize hardware and software configuration information.
2. Express assignment, data flow, and scale information algorithmically in a rich and natural manner.
3. Support scaling of heterogeneous system components.
4. Dont impose an application design model.
5. Remain independent of system-specific APIs.
6. Eliminate all target-specific setup and initialization code.
7. Enable fast turnaround of configuration changes.
8. Support deployment (no development tools / workstations).

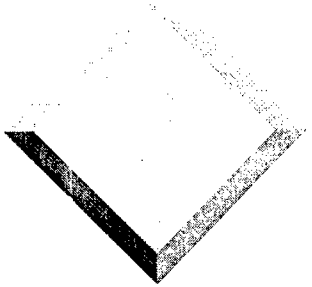
The Talaris Application Framework

The Generator - application development

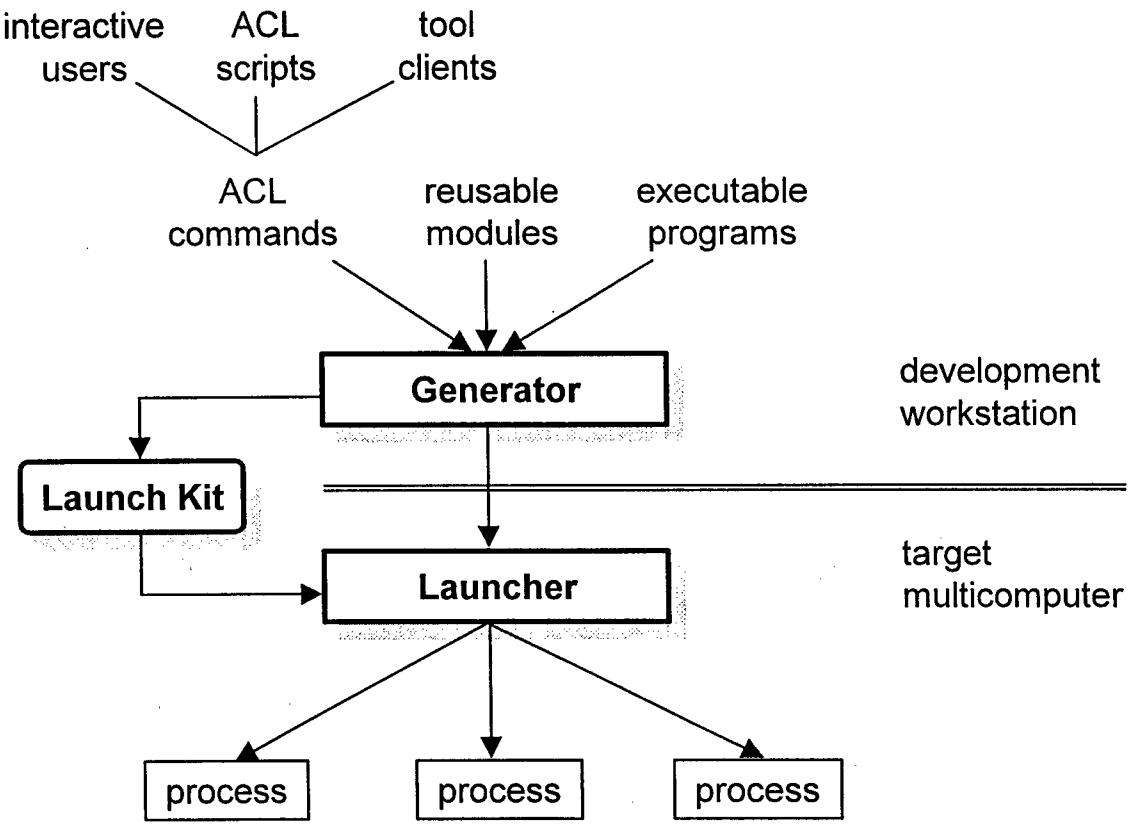
- runs on the workstation
- processes the Application Configuration Language (ACL)
- accepts software components and instructions for their use
- creates a Launch Kit

The Launcher - initialization and execution

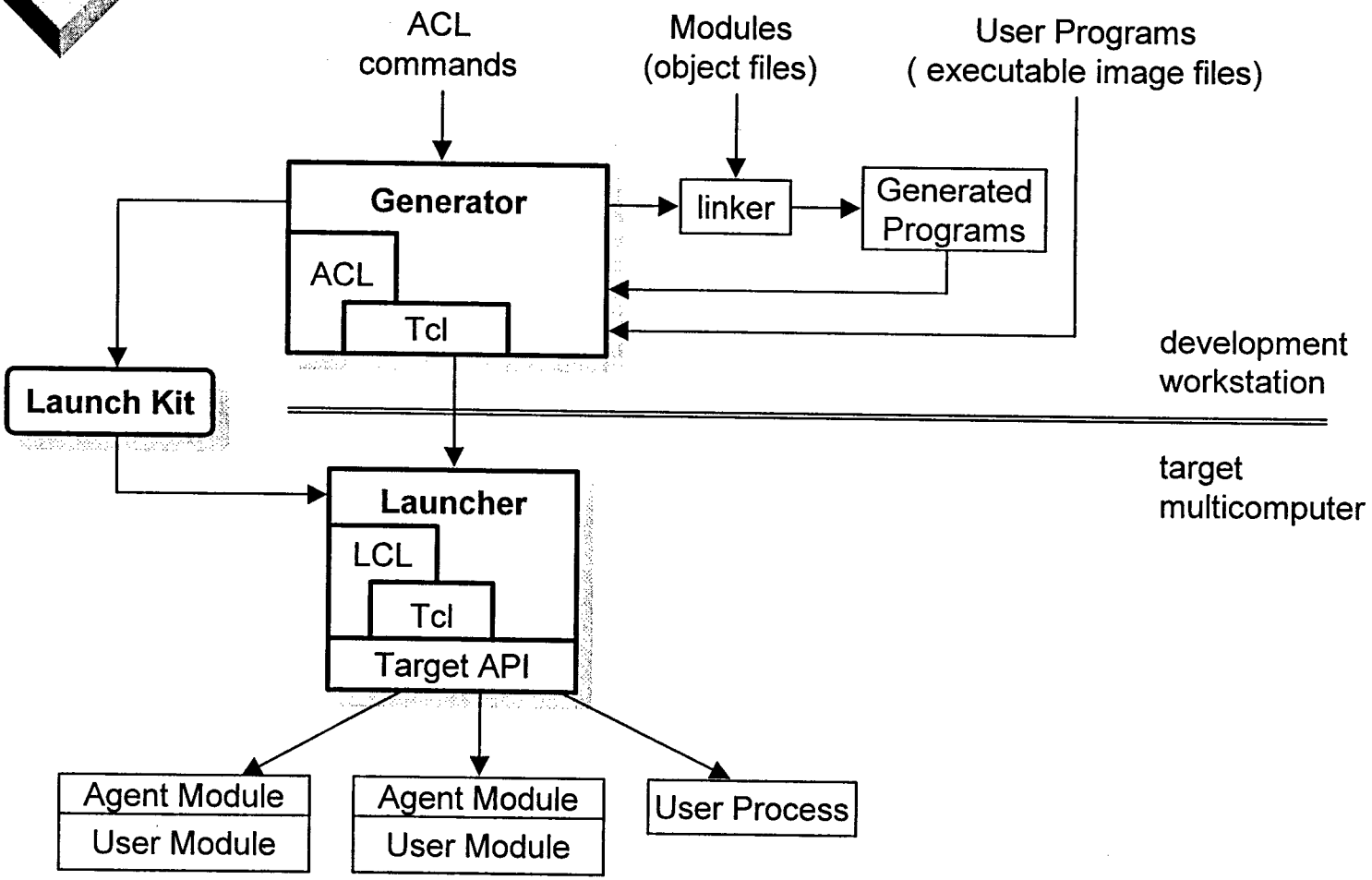
- runs on the target
- is integrated with the target's shell tool, if any
- accepts a Launch Kit
- creates running applications

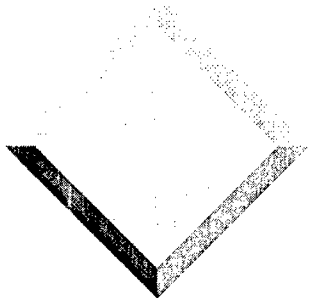


Talaris Subsystems

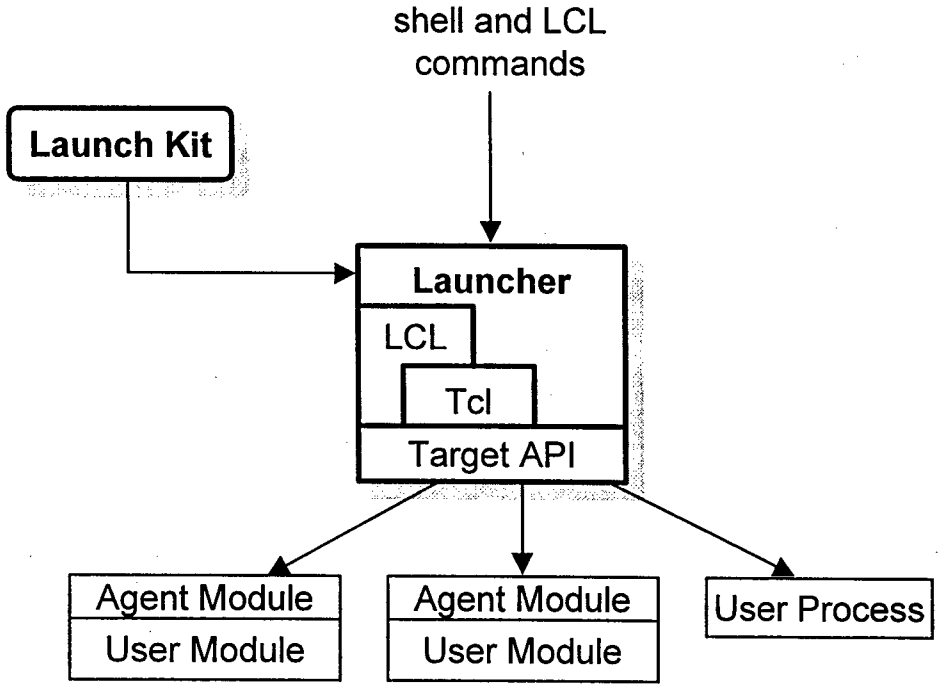


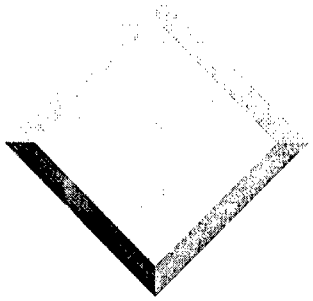
Talaris Subsystems - Expanded





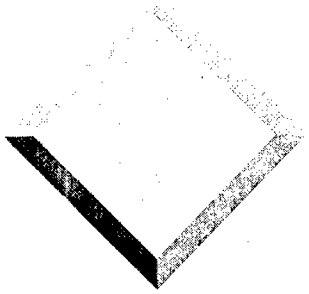
Deployment



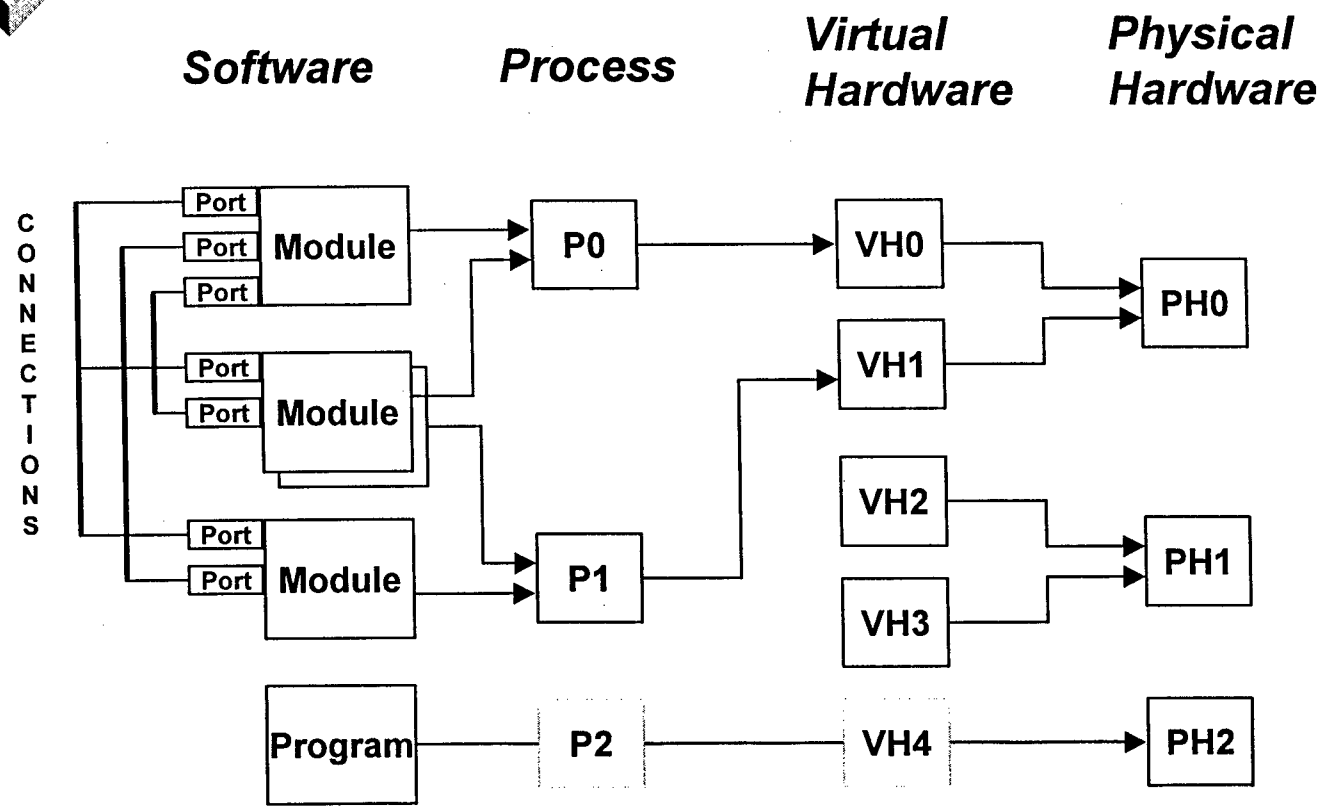


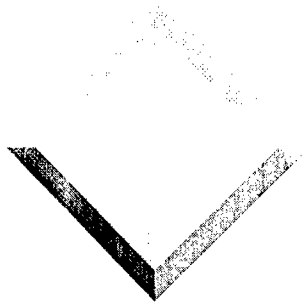
The Talaris Difference

CONVENTIONAL DEVELOPMENT	TALARIS APPLICATION FRAMEWORK	
create/port application-specific code	**SAME**	*developers focus on the application*
create makefile for compilation	**SAME**	*Talaris accepts compiled code*
learn system-specific APIs	n/a	*Modules can use generic mechanisms*
create/debug setup code	**ACL**	*developer specifies connections*
add to makefile for compilation	n/a	*setup is data-driven*
create makefile for building executables	**ACL**	*developer specifies assignments*
build and organize executables	n/a	*performed by Generator*
create/debug run-time scripts/code	n/a	*internal to Launcher*
debug and tuning	**SAME**	*platform-specific tools*



The Application Model





Basic Concepts

Entities

Components perform processing, are synthesized as needed

- software and hardware

Ports are interfaces to other Components

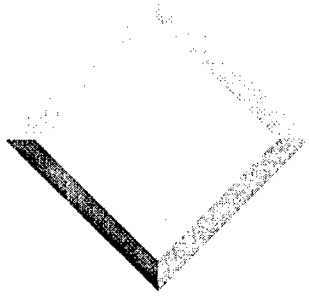
- semaphore, socket, shared memory

Connections provide pathways to other Components'Ports.

- data transfer, data sharing, synchronization
- are full-fledged objects

Domains and Assignment

Software: reusable Modules or legacy Programs
Process: a collection of modules or a Program
Virtual hardware: the ideal hardware configuration
Physical hardware: the actual hardware configuration
Components are assigned across Domains, can skip
Assignment implies aggregation



Step 1: Describe the Application

Develop the inventory of software components

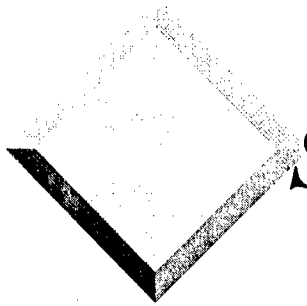
- User Modules: conforming to the Talaris Module model
- User Programs: separately created executable images

Describe the application in an ACL script

- create Module and Program objects
- connect the Modules'Ports
- create Processes if needed
- create the Virtual and Physical Hardware objects
- specify assignments

Load the ACL script into the Generator

- basic error-checking during command processing
- interactively inspect the application



Step 2: Generate the Launch Kit

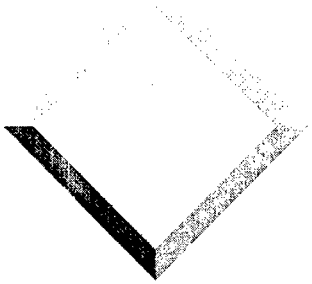
The Generator

analyzes the application
finalizes assignments
performs additional validation
devises a plan for initializing connections
creates Generated Programs from User Modules and Agent Module
creates the Launch Configuration File

The Launch Kit

Launch Configuration File
executable images for

- Generated Programs (made from User and Agent Modules)
- User Programs (not created by the Generator)



Step 3: Launch the Application

The Launcher

- analyzes the Launch Kit
- extracts and executes the launch script
- sets up global IPCs
- loads images, spawns processes

The Agent Module

- a special Module provided with the framework
- linked into the executable images for Generated Programs
- provides the main( ) entry point

The Agent at run-time

- uses data created by the Generator
- creates and initializes local IPC's
- gets Modules ready to run, reports back to Launcher
- starts the Modules when directed by Launcher

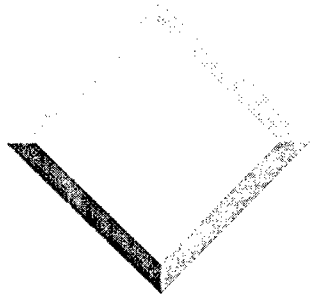
Application Configuration Language

The Generator's ACL interpreter

- extends the standard Tcl interpreter - "generic" Tcl commands
- incorporates the "expect" extension commands
- provides immediate error checking and feedback
- supports interactive query of the current application configuration

ACL commands

types	declare, get_type, delete
instances	create, delete
properties	set_property, get_property, delete_property
scaling	set_scale, get_scale
assignments	assign, deassign
connections	connect, disconnect
information	query
control	generate, load, setup, start, run



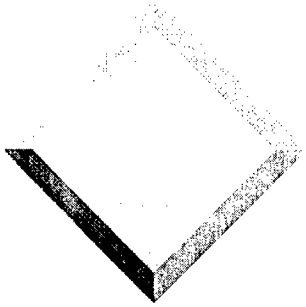
Tool Command Language

Why based on Tcl?

easy to learn, extensively documented, books available
vs. UNIX shells: portable, extensible, text/list oriented
vs. Perl: small, embeddable, tunable

Tcl commands (partial list)

variables	set, $x, array, $x(y), incr, argv, env
control	if, for, foreach, while, exit
lists	list, lappend, llength, lindex, lreplace, concat
strings	string, join, split, append, format
functions	proc, return
. . .	puts, catch, eval, exec, expr, trace



Tcl Example

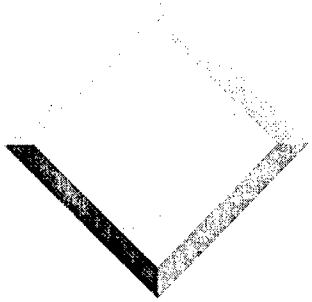
```
set date [exec date]
foreach file $argv {
    set fileId [open $file]
    set lineNumber 1
    puts "\nFile: $file\nDate: $date\n"
    puts "Line\tLength\tData\n"
    while { ! [eof $fileId] } {
          gets $fileId line
          puts "$lineNumber\t[string length $line]\t$line"
          incr lineNumber
    }
    close $file
}
```

```
% list-file /tmp/a-sample-file

File: /tmp/a-sample-file
Date: Sun Sep 10 12:55:47 EDT 1995

Line      Length    Data

1         11        First line.
2         13        Another line.
3         10        Last line.
```

ACL Types

All objects are typed

All types have the base type "Object"

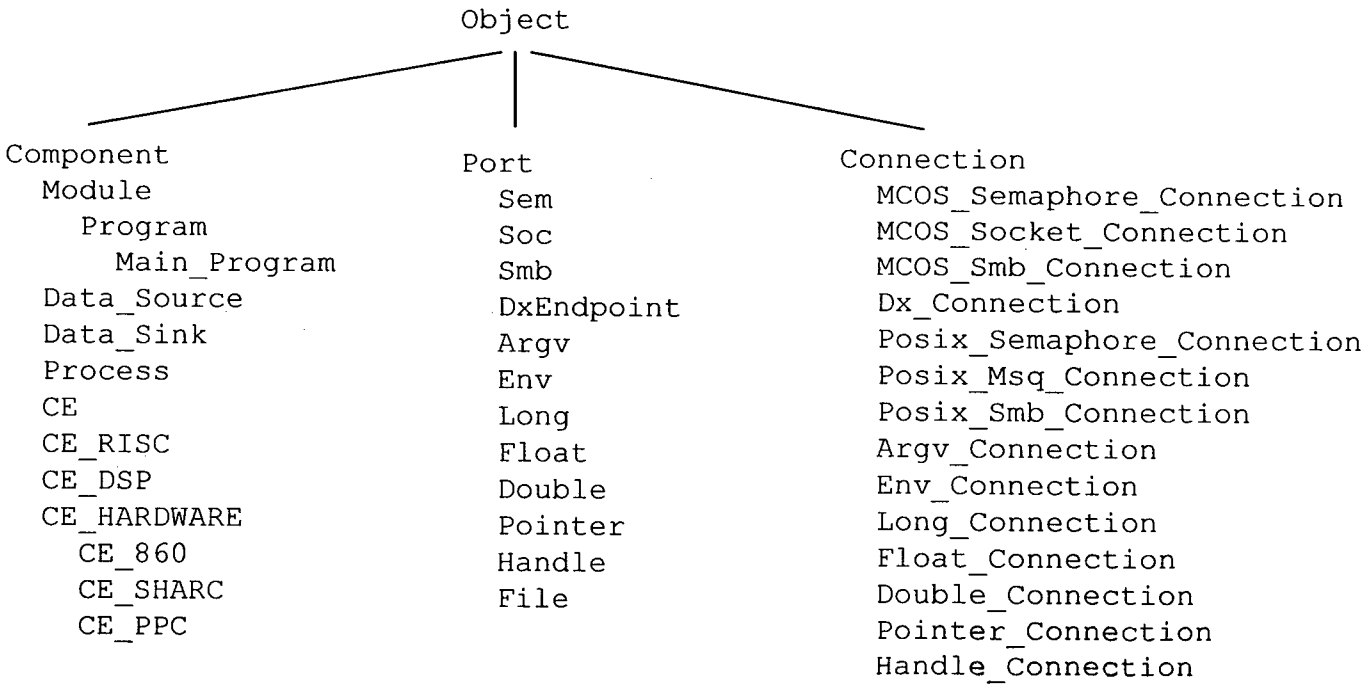
New types are derived with "declare" command

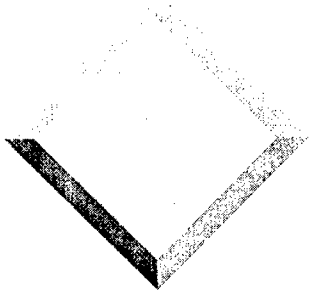
```
declare Module FFT {Sem sync_in Sem sync_out Smb in Smb out}
```

ACL predefines many intrinsic types

```
intrinsics.acl
```

Intrinsic Types (partial list)





ACL Objects

"create" produces objects

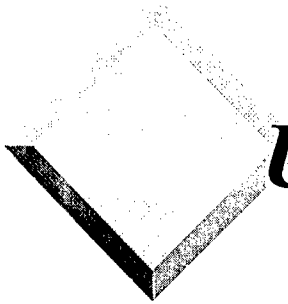
```
create Program hello
```

Objects can be scaled (and re-scaled)

```
create Program hello<8>
create CE_860 front-end<64>
```

Scaled objects can be assigned

```
set n 4
declare Program RadarInput -file ./examples/radar_in
create RadarInput input<$n>
declare CE_860 RadarInterface -memory 16MB
create RadarInterface interface<$n> -ceid 27
assign input -to interface
```

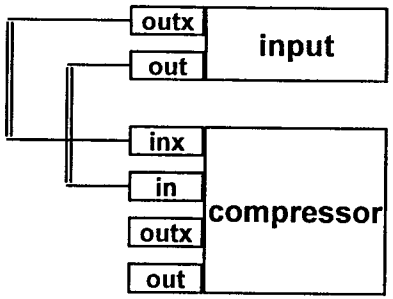



Using Modules in an Application

Module: simply a logical functional unit

- well-defined interfaces and processing
- each Module runs in its own thread
- granularity is up to the designer/developer
- IPC mechanisms are already set up
- subsequent call depends on the return value

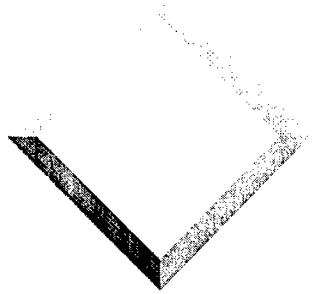
Using Modules in ACL




```
declare Module FrameInput {Sem outx Smb out}
create FrameInput input

declare Module FrameCompressor {Sem inx Smb in Sem outx Smb out}
create FrameCompressor compressor

connect input.outx compressor.inx
connect input.out compressor.in
```

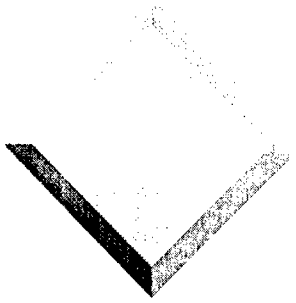
Designing a Module

Module entry point

- C calling conventions
- function signature is described in ACL
- function parameters correspond to Ports -- single or vectors
- Port parameters are directly usable with the native operating system API

Module implementation

- written in C or other supported HLL
- reentrant
- careful use of global variables, if at all
- compiled, optionally placed in archive (*.a)



Coding and Using a Module

Source: frame_input.c

```
#include "talaris.h"
int input ( T_Sem outx, T_Smb out )
{
    /* transfer input from device to 'out'          */
    /* give semaphore 'outx'                        */
    return 0;           /* OK to call continuously */
}
```

Compilation

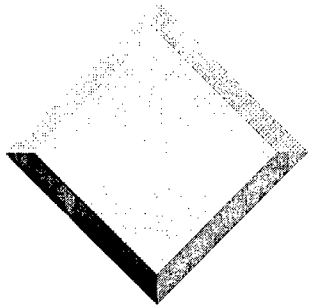
```
cc -c frame_input.c
```

Description

```
declare Module FrameInput {Sem outx Smb out}
```

Application usage

```
create FrameInput input -file input.o
```



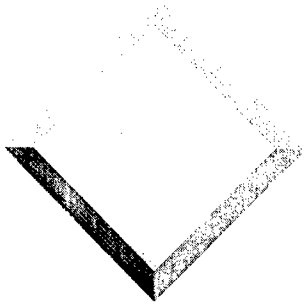
Generated Programs

Generated Program = Agent Module + User Module(s)

- Assignment results in Module aggregation
- the Generator automatically creates Processes as needed
- one copy of the code for each Module
- function names & symbols are preserved, source debuggers still work

The Agent Module

- is the main() for the Generated Program
- decodes its Agent script (created by the Launcher)
- creates and initializes local IPC's
- creates a thread for each Module
- sets up the call to each Module entry point
- synchronizes entry of Modules



User Programs

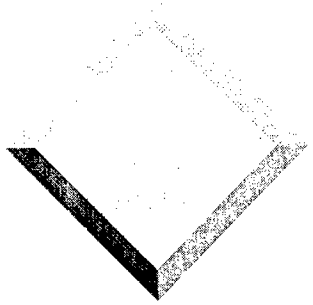
User Programs are existing executables

ACL is still used to assign and run them

```
create Main_Program display -argv {-d 2 -p 512 gdx1c}
create CE_RISC video_output -ceid 17
assign display -to video_output
```

Included in the Launch Kit by the Generator

Applications often mix User Programs and Generated Programs



For Developers...

"Fewer moving parts" - less setup code, makefiles, scripts

The natural application design is intact

Adoption of the Module approach is not forced

Modules can use native system APIs

No generated code is added to the application

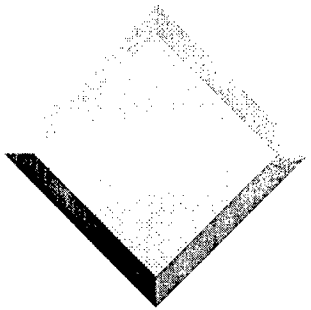
No run-time performance impact

Each generated executable is tailored

Debug with existing tools

Quick turnaround by a data-driven Launcher

Works with configuration management tools



For Tool Clients...

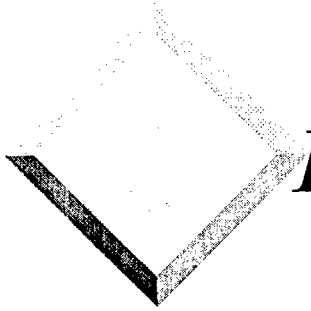
A "tool-friendly" system software environment

Fewer system-specific details that slow down tool development

Multiple platforms give tools more leverage

Tool developers can focus on their added value

Each tools can keep its unique application / design focus



Further Topics and Information

"ACL Specification" - Mercury Computer Systems, Inc.

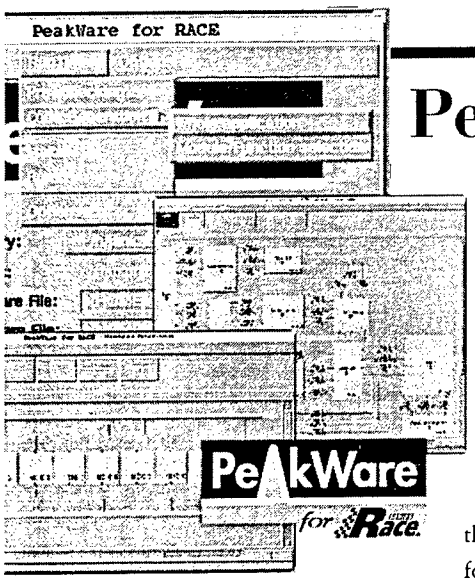
- scaled Ports
- undo of assignments, connections
- optimized rebuild / relaunch
- Properties and Attributes
- configuring the Generator
- system-specific commands
- identifiers and references
- Port name finalization
- query of the configuration
- changing scale
- writing re-scalable modules
- C interfaces to the ACL database
- Launch Kit contents
- Launch Configuration Language
- scaleless connections
- resolving connections

"Tcl and the Tk Toolkit" - John Ousterhout (Addison Wesley)
"Practical Programming in Tcl and Tk" - Brent Welch (Prentice Hall)
"Exploring Expect" - Don Libes (ORielly & Associates)

Appendix B.

Talaris applied to Peakware product

(Mercury product announcement)



PeakWare™ for RACE®

Computer Systems, Inc.
MERCURY

The Component Programming Development Environment for Embedded Applications

Create Deployable Application Code for Embedded Multicomputer Systems

Reduce Software Application Life-Cycle Cost

Comprehensive, Component-Based Development Environment

PeakWare for RACE, by Mercury Computer Systems, Inc., refines the concept of stream computing by applying to it the most productive application development interface in the industry. It is a fully graphical tool for designing and deploying applications for the RACE multicomputing environment. PeakWare for RACE employs a building-block process that provides a logical, intuitive development environment familiar to anyone who has ever sketched out a system design on a white board.

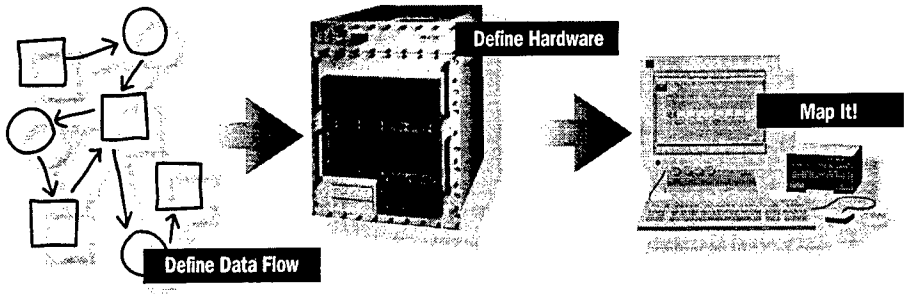
With PeakWare for RACE, programmers can develop signal processing applications without having to rewrite existing, proven algorithms, and configure hardware components without worrying about whether different processors will communicate with the software. PeakWare for RACE's visual representation of application data flow lets users access distinct software domains, hardware configurations, and mapping of software modules to hardware modules through simple, efficient graphical representations. Using PeakWare for RACE, programmers can change hardware resources and configurations without having to rewrite source code. With its simple point-and-click functionality, code is generated, applications are created, and easy-to-follow graphical documentation gets produced.

PeakWare for RACE

An intuitive graphical user interface (GUI) enables application developers to depict software modules and the intended connection between the modules and target processors. PeakWare for RACE allows the user to easily map software modules to target hardware, providing an unprecedented level of productivity and portability without hampering performance.

Productivity

With PeakWare for RACE, software and hardware domains remain uniformly distinct, allowing different hardware configurations to be used without requiring changes to the source code. With software components that are wholly independent of the hardware configuration, due to



Race SERIES

automatic source code generation for configuration-dependent communications code, application developers can seamlessly upgrade processors or insert new technology. Engineering productivity is dramatically improved, and time-to-market is substantially reduced as developers can spend less time rewriting code and more time streamlining a system for optimal performance.

PeakWare for RACE contains extensive turnkey code libraries, and also allows developers to easily incorporate their own code, either in source code format or through a feature called Opaque Modules in compiled (object) format. Because this innovative tool keeps functional code separate from platform-specific code, it facilitates software reuse.

Portability

PeakWare for RACE allows developers to use different hardware maps for any combination of processors that may already exist within a RACE multicomputer system. Furthermore, application code can be targeted at different system and backplane architectures. With PeakWare for RACE's ability to mix high-performance processors such as the PowerPC™, SHARC®, and i860 in heterogeneous multicomputer configurations, developers can test how well their individual algorithms, as well as complete applications, work in each case in order to create the optimal performance match.

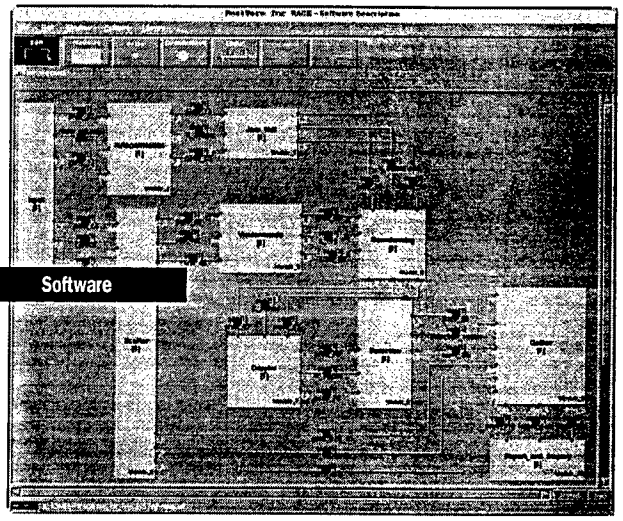
Performance

PeakWare for RACE was created from the ground up to reduce development costs and time-to-market, and is the only development tool that generates full-performance, production-ready, deployable code that is ready to run on targeted configurations. Speed, accuracy, and ease of use have been key criteria for this tool every step of the way, making PeakWare for RACE a true competitive advantage.

With PeakWare for RACE, a developer graphically creates an application in three parts. The first element, software design, allows the developer to define the interconnection of software components. In the next phase, the RACE system hardware configuration is graphically defined. Finally, the developer maps the interconnection design to the target hardware platform.

Software Design

For pure software design, developers use the PeakWare for RACE GUI to establish data-flow communications



between various software modules and their related protocols. The software graph enables programmers to create a graphical representation of the application they are developing. Programmers have access to six main graphical operations: selecting the design options, inserting the module component in the graph, defining the module contents, inserting the function in the module graph, defining the function, and saving the software graph.

The software graph handles only information about software processes and communication declarations. For the software part of the specification, the developer needs minimal knowledge of the target hardware.

Because the software graph's basic building block is a function, users can either create their own functions or manipulate those provided with PeakWare for RACE. Either can be retrieved from libraries within the application.

Another key component in the software design is a module. A module is always implemented as a process or a thread (task) and is composed of a function or a set of functions. PeakWare for RACE offers a graphical module

editor to describe the internal structure of modules, functions, and data exchanged between functions. At this level, the application developer can interconnect modules either directly or using a connection such as shared memory buffers, sockets, or semaphores.

PeakWare for RACE allows an application developer to easily create modules and their links. The ports by which a module interconnects to other modules are defined using the GUI to select a port type from those types supported (e.g., shared memory with DMA access, or socket, etc.). Each software graphical object (module, connection, or function) has a description window, depending on the object type, which lets the developer override default settings.

The application developer can also edit graphical objects and move through the software module's hierarchy. PeakWare for RACE provides Top, Down, and Up buttons and navigation menu options as well as a hierarchy display window for users to navigate through the module's hierarchy and contents.

A scaling factor can be applied to software modules and other graphical objects to generate a specific number of fully intercommunicating and synchronized source code instantiations. For example, scaling a module to 32 results in source code generation for communication and synchronization of the 32 instances of the module. Regardless of the amount of scaling, the code structure of the module is compiled once in PeakWare for RACE's library. A software description field gives programmers a place to define custom functions or annotate PeakWare for RACE functions.

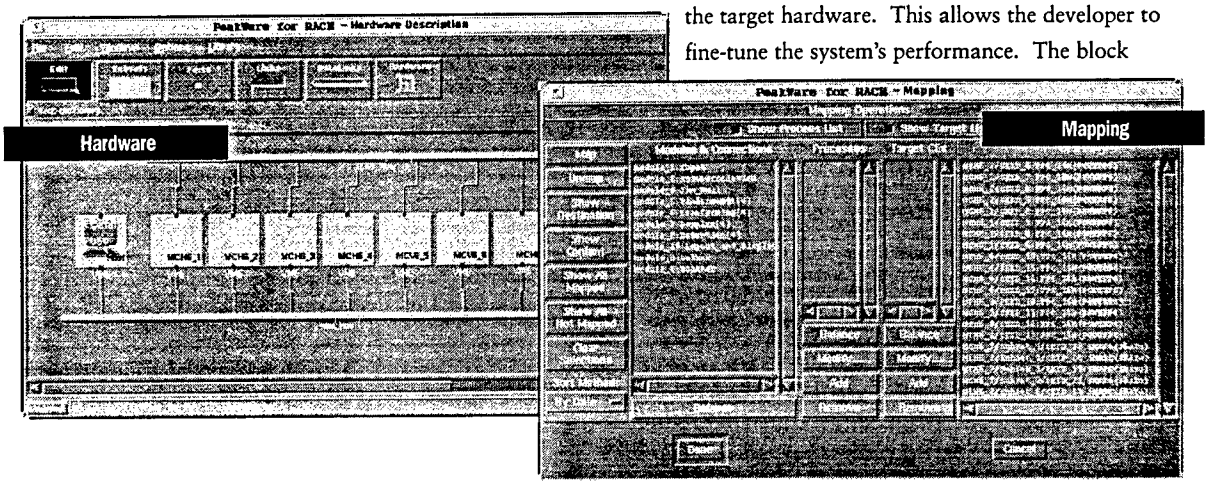
Hardware

In keeping with the ease of component programming, PeakWare for RACE gives developers six main functions for GUI-based hardware system definition: creating the graph components, defining the hardware configurations, defining the data links, linking the components, specifying the application host or the targeted hardware, and saving the hardware graph. With these functions, application developers graphically describe the RACE multicomputer configuration that is available or needed to accommodate the real-time performance requirements of the application software.

The target configuration can be graphically displayed, modified (e.g., adding a hardware board), or created from elementary, predefined hardware components, such as specific boards (RACE family of high-end signal processing boards or SPARC Unix host boards) and interconnects (RACEway crossbar or VMEbus). PeakWare for RACE's turnkey library of hardware components ranges from the simple to the complex. In addition, developers can add their own graphically defined hardware configurations to the library. Furthermore, PeakWare for RACE gives programmers a simple, flexible, and efficient way to test hardware components by allowing changes in hardware configurations — even adding or subtracting processors — without rewriting or editing source code.

Mapping

Once the software and hardware are defined, the developer can control the way the application is mapped onto the target hardware. This allows the developer to fine-tune the system's performance. The block



diagram, data-flow design metaphor extends to PeakWare for RACE's mapping capabilities, allowing users to assign particular software modules to specific processors or other pieces of hardware. The developer can map modules onto processors with the Mapping operation, and produce information that is then used by the Code Generator.

Each module results in a thread at run time. With the Mapping window, the developer can override the default mapping of modules (threads) into processes, or the default mapping of virtual hardware onto physical processors. A single software module may be assigned to one or more processors, or many software modules to one processor. This feature encourages the programmer to focus on streamlining for deployable application performance, and eliminates the worry of software and hardware communication.

In Partnership

PeakWare for RACE is the result of collaboration between Mercury Computer Systems and MATRA SYSTEMES & INFORMATION (Matra MS&I), a France-based industry leader in the development and integration of high-performance computing solutions.

System Requirements

Development Host

- SPARC system running Solaris™ 2.4 or 2.5
- Must have X display terminal capability (color recommended)
- 75 MB for complete on-line help with screen dumps
- 16 MB memory minimum (32 MB recommended)

Runtime Host

- Any Mercury-supported runtime host

Mercury Hardware/Software

- MC/OS Development and Runtime Environment Version 4.3 and later
- SHARC-, PowerPC-, or i860-based system with minimum 8 MB memory per node

Shipping Media

- 1/4 inch QIC-150 tape, pkgadd format
- 8 mm tape, pkgadd format

MATRA SYSTEMES & INFORMATION

Computer Systems, Inc.
MERCURY
The Ultimate Performance Machine

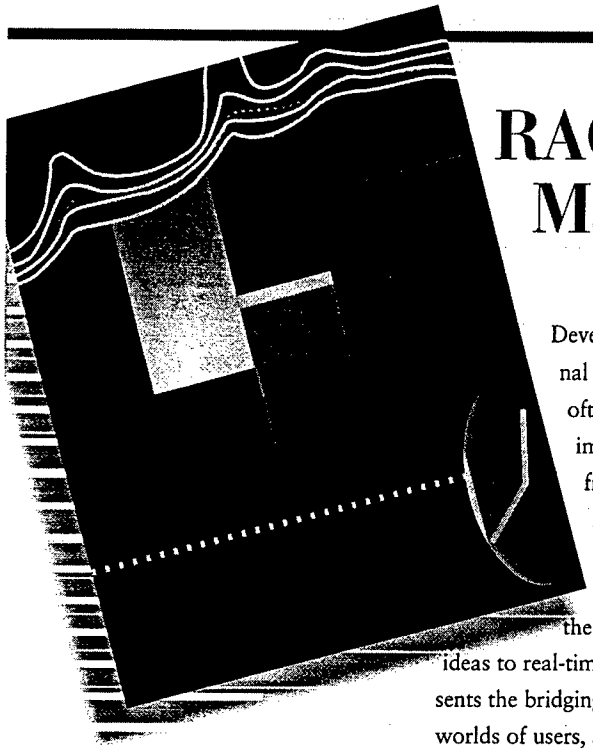
199 Riverneck Road
Chelmsford, MA 01824-2820 U.S.A.
978-256-1300 · Fax 978-256-3599
800-229-2006 · http://www.mc.com
NASDAQ: MRCY

DS-5U-10

Appendix C.

Further product developments towards MATLAB use

(Mercury product announcement)



Computer Systems, Inc.
MERCURY

RACE® MATLAB® Math Library

Easily Deploy MATLAB Designs to Embedded, Multi-Node Computing Systems

Automatically Convert M-Files to C Code for RACE Systems

Increase Overall Developer Productivity

Developers of demanding signal processing applications often face the challenge of implementing a prototype from software which originated during the research of an idea or concept. For many organizations, the migration of these new ideas to real-time proof-of-concept represents the bridging of two vastly different worlds of users, systems, and software methodologies.

The RACE MATLAB Math Library enables M-file programs developed in The MathWorks' powerful MATLAB environment to be rapidly targeted to RACE i860 and PowerPC™ embedded computer systems.

Developed by Mercury Computer Systems, a producer of high-performance embedded multicomputing systems, and The MathWorks, a developer of high-performance numeric computation software, the RACE MATLAB Math Library consists of the RACE Embedded MATLAB Math Library and the RACE Development MATLAB Math Library. In conjunction with The MathWorks' MATLAB Compiler, the embedded library allows application execution on the i860- and PowerPC-based RACE systems, and the development library provides single-precision, MATLAB-compiled M-file execution on Sun™ Solaris™ workstations.

Mercury's unique RACE MATLAB Math Library offers a new way to reduce time-to-prototype for high-performance digital signal processing projects. It can eliminate from weeks to months off project development schedules, and eases the task of going from workstation-based research to a real-time, high-performance embedded solution.

- The RACE Embedded MATLAB Math Library allows MATLAB M-files to be compiled and executed on multiprocessor RACE Series i860 and PowerPC compute nodes. This eliminates the need to manually convert M-files to C code.
- Once a MATLAB application is implemented on the RACE system, life-cycle support and functional evolution is greatly facilitated. In the past, it was not possible to avoid the costly and unmanageable practice of having two divergent code bases — one for the researcher and one for the developer — as code refinements and newer ideas emerge from actual prototype data. By using MATLAB as a "common language," researchers work with development engineers on the real-time system, saving significant time for fast-evolving programs.

- Real-time implementations often require single-precision math. Until now, there was no way to use MATLAB, which is a double-precision tool, to effectively test the effect of reducing precision. The effects of precision can force different algorithm strategies, causing delays in obtaining a prototype system. The RACE MATLAB Math Library minimizes recoding for real-time prototyping and reveals effects of single-precision at the research phase through the use of the RACE MATLAB Development Math Library.
- When used with multicomputing techniques, the RACE MATLAB Math Library provides a higher-performance platform to accelerate MATLAB project evaluation. While workstation implementations are limited by the performance of a single workstation, the RACE MATLAB Math Library provides for multiple processor implementations.

Targeting MATLAB M-Files to the RACE Multicomputer

When used in conjunction with the MATLAB Compiler, the RACE Embedded MATLAB Math Library allows MATLAB M-files to compile for a RACE i860 or PowerPC compute environment. The resulting executable is single-threaded.

The RACE MATLAB Math Library is intended for early prototype systems transitioning from research to deployment. Improved performance over workstations is accomplished through a multicomputer implementation. When moving from prototype to deployment, performance-critical regions may be tuned by directly utilizing Mercury's optimized Scientific Algorithm Library (SAL) or by recoding to other processors such as the SHARC® DSP.

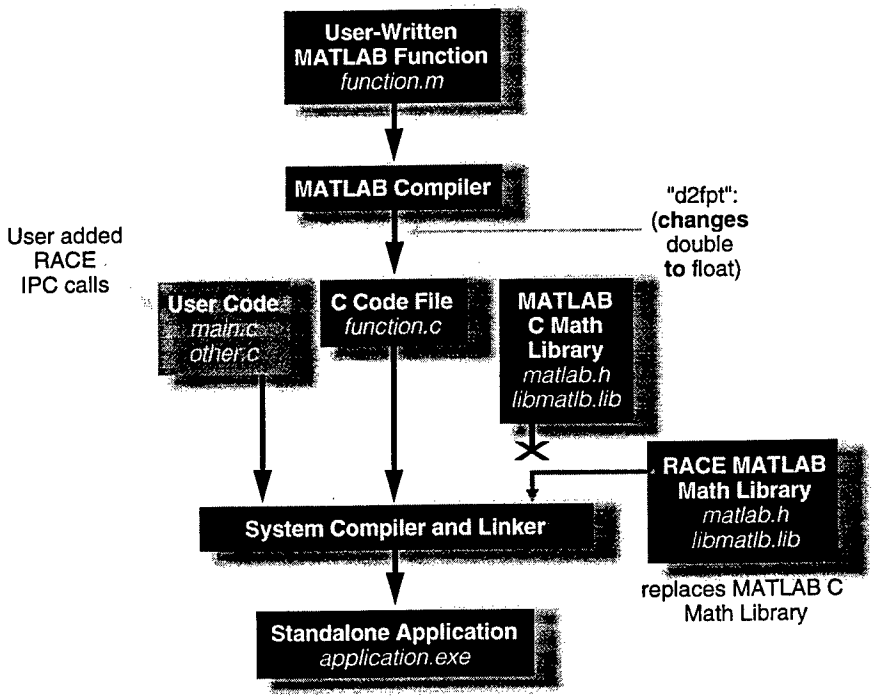
Assuming that a "monolithic" M-file exists for an application, the following steps are required to implement a RACE multicomputer MATLAB program:

1. A multicomputing strategy is developed for partitioning processing across multiple processors.

2. Using the partitioning strategy as a guide, the monolithic M-file is carefully studied and re-implemented in MATLAB as multiple separate M-files that each represent a piece of useful processing.

3. Each M-file is compiled with the MATLAB Compiler to create a MATLAB "component" which is a C callable function. For purposes of the programming model, the developer can essentially think of these MATLAB components as their own C callable SAL-like functions.

4. To achieve a multicomputer application, the developer must implement the interprocess communication (IPC) to



provide data movement and synchronization to logically interconnect these MATLAB components. For RACE multicomputers, there are several choices for IPC, including shared memory buffers with semaphores and Mercury's Parallel Application System (PAS™).

The PAS application comprises a high-performance set of libraries which forms a complete programming environment for developing parallel applications in a distributed memory multicomputer system while maintaining maximum hardware performance.

5. The application code is compiled and linked against required libraries, one of which is the RACE Embedded MATLAB Math Library, to create executables. The choice of i860 or PowerPC, and single or double precision is made here by selecting the desired compiler and library names. Launching and debugging of the application is accomplished using standard RACE development tools for C applications (see "Space-Time Adaptive Processing Using the RACE® MATLAB® Math Library," AN-5C-10).

The MATLAB M-files on RACE are subject to the limitations of the MATLAB Compiler, such as the inability to display graphics. Typically, output data is sent to the development workstation or written to disk and displayed with MATLAB executing on the workstation.

RACE MATLAB Math Library Product Description

The RACE Embedded MATLAB Math Library consists of a user's guide, detailed code examples, help-line support, and four libraries each for the i860 and PowerPC. The RACE Development MATLAB Math Library consists of two libraries for the Sun SPARC® workstation.

	Platform	
RACE MATLAB Math Library Support	**Embedded**	**Development**
Single-Precision	X	X
Double-Precision	X	X
Vectorized Single-Precision	X	
Vectorized Double-Precision	X	

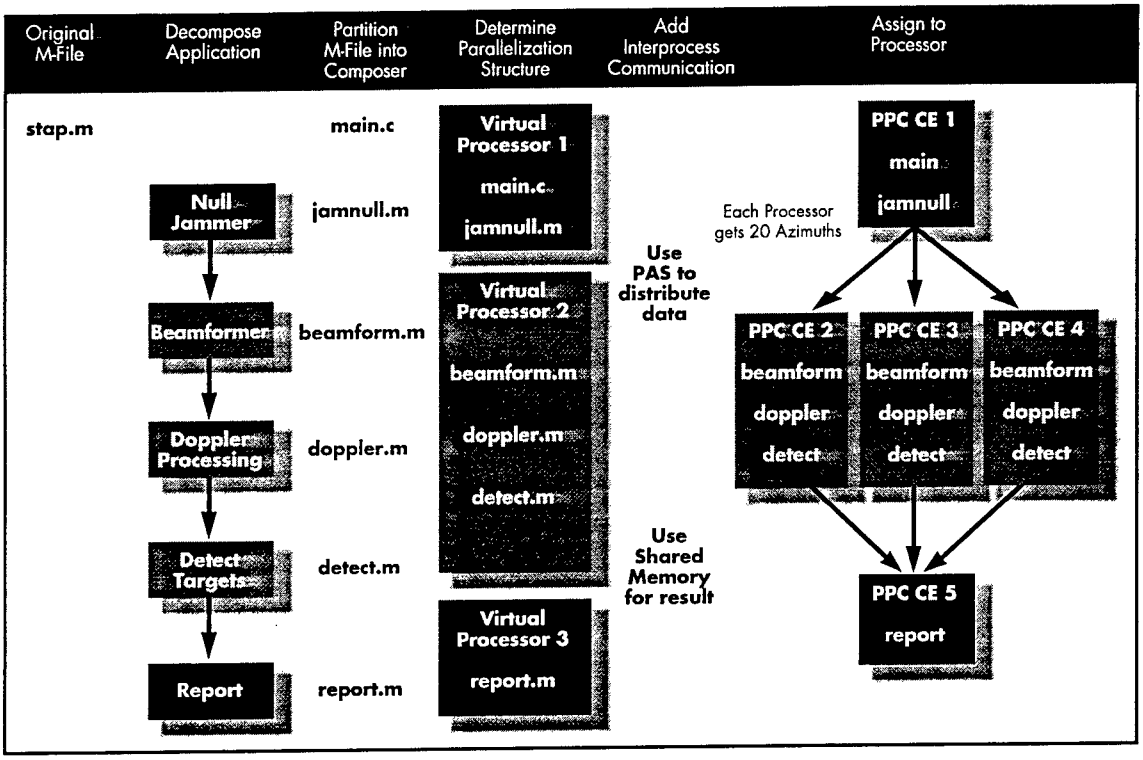
NOTE: Due to processor architecture differences, double-precision results may not match exactly between the workstation, and i860 and PowerPC compute nodes. The embedded optimized library vectorizes MATLAB routines with routines from Mercury's SAL for higher performance. Since algorithm implementation methods can impact precision, both optimized and non-optimized libraries are available to detect such influences.

RACE MATLAB Math Library Product Example

To illustrate the power of the RACE MATLAB Math Library, a fully documented example of space-time adaptive processing (STAP) radar is included in the product (see "Space-Time Adaptive Processing Using the RACE® MATLAB® Math Library," AN-5C-10).

The original STAP M-file applies mathematical computations to radar data for ultimate target detection. Hand-coded C routines distribute the processing among multiple processors on the RACE target system by applying a PAS multiprocessor master/slave model. The STAP Application Example (see page 4) shows a graphical model of this multicomputing adoption of a linear mathematical MATLAB application.

MATLAB is a powerful tool for application development in defense signal processing and diagnostic medical image reconstruction. With the RACE MATLAB Math Library, researchers and development engineers can speed the transition from the "drawing board" to the "product shelf."



STAP Application Example

Order Information, for RACE MC/OS™ Version 4.x*

Item	Part Number
V1.1.0 Bundled - RACE MATLAB Math Library(1)	810-07103
V1.1.0 Embedded - RACE MATLAB Embedded Math Library	810-07101
V1.1.0 Development - RACE MATLAB Development Math Library(2)	810-07102

(1) Includes both Embedded and Development Library

(2) Requires purchase of Embedded Library

*The RACE MATLAB Math Library is supported on MC/OS v4.2 and later

Compatibility with MATLAB Products

V1.1.0 MATLAB 5.2/1.2i860, PowerPC available now, SHARC n/a

MATLAB PARTNER

•U.S. GOVERNMENT PRINTING OFFICE: 1999-510-079-81212

The MathWorks, Inc. is the leading developer and supplier of technical computing software worldwide. More than 400,000 technical professionals, educators, and students in more than 100 countries use The MathWorks' MATLAB® interactive computational language, math, and visualization tool. Founded in 1984, The MathWorks is a privately held company located in Natick, Massachusetts.

Computer Systems, Inc.
MERCURY
The Ultimate Performance Machine

199 Riverneck Road
Chelmsford, MA 01824-2820 U.S.A.
978-256-1300 • Fax 978-256-3599
800-229-2006 • http://www.mc.com
NASDAQ: MRCY

DS-5T-11

AFRL/IFTC
ATTN: RALPH KOHLER
26 ELECTRONIC PKWY
ROME NY 13441-4514 2

MERCURY COMPUTER SYSTEMS, INC
199 RIVERNECK ROAD
CHELMSFORD MA 01824-2820 2

AFRL/IFOIL
TECHNICAL LIBRARY
26 ELECTRONIC PKY
ROME NY 13441-4514 1

ATTENTION: DTIC-OCC
DEFENSE TECHNICAL INFO CENTER
8725 JOHN J. KINGMAN ROAD, STE 0944
FT. BELVOIR, VA 22060-6218 2

DEFENSE ADVANCED RESEARCH
PROJECTS AGENCY
3701 NORTH FAIRFAX DRIVE
ARLINGTON VA 22203-1714 1

ATTN: NAN PFRIMMER
IIT RESEARCH INSTITUTE
201 MILL ST.
ROME, NY 13440 1

AFIT ACADEMIC LIBRARY
AFIT/LDR, 2950 P.STREET
AREA B, BLDG 642
WRIGHT-PATTERSON AFB OH 45433-7765 1

AFRL/MLME
2977 P STREET, STE 6
WRIGHT-PATTERSON AFB OH 45433-7739 1

AFRL/HESC-TDC
2698 G STREET, BLDG 190
WRIGHT-PATTERSON AFB OH 45433-7604 1

ATTN: SMDC IM PL
US ARMY SPACE & MISSILE DEF CMD
P.O. BOX 1500
HUNTSVILLE AL 35807-3801 1

TECHNICAL LIBRARY D0274(PL-TS) 1
SPAWARSYSCEN
53560 HULL ST.
SAN DIEGO CA 92152-5001

CDR, US ARMY AVIATION & MISSILE CMD 2
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSAM-RD-OB-R, (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5000

REPORT LIBRARY 1
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545

ATTN: D'BORAH HART 1
AVIATION BRANCH SVC 122.10
FOB10A, RM 931
800 INDEPENDENCE AVE, SW
WASHINGTON DC 20591

AFIWC/MSY 1
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016

ATTN: KAROLA M. YOURISON 1
SOFTWARE ENGINEERING INSTITUTE
4500 FIFTH AVENUE
PITTSBURGH PA 15213

USAF/AIR FORCE RESEARCH LABORATORY 1
AFRL/VSOSA(LIBRARY-BLDG 1103)
5 WRIGHT DRIVE
HANSCOM AFB MA 01731-3004

ATTN: EILEEN LADUKE/D460 1
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730

OUSD(P)/DTSA/DUTD 1
ATTN: PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

*Total Number of Copies is: 24

MISSION OF AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.